A Survey and Empirical Comparison of Modern Pseudo-Random Number Generators for Distributed Stochastic Simulations

Marcus Schoo Krzysztof Pawlikowski

Department of Computer Science and Software Engineering

and Donald C. McNickle

Department of Management

University of Canterbury
Christchurch, New Zealand

TR-CSSE 03/05

Abstract

Distributed stochastic simulations has become a popular tool for evaluating and testing complex stochastic dynamic systems. However there is some concern about the credibility of the final results of such simulation studies [11]. One of the important issues which need to be properly addressed for ensuring validity of the final results from any simulation study is application of an appropriate source of randomness. In the case of distributed stochastic simulation, the quality of the final results is highly dependent on the underlying parallel Pseudo-Random Number Generator (PRNG).

Parallel PRNGs (PPRNGs) with the required empirical, analytical and deterministic properties are not trivial to find [9, 23, 10]. However, much research has resulted in several generators which we consider to be of high quality [6, 23, 24, 28, 32]. The effectiveness of simulations however depends not only on their accuracy but also on their efficiency and so simulations are also reliant on the speed and flexibility of these PPRNGs.

In this paper, without a loss of generality, we examine the required features of modern PPRNGs from the point of view of their possible applications in Multiple Replications in Parallel (MRIP) paradigm of stochastic simulation. Having surveyed the most recommended generators of this class, we test their implementations in C and C++. The generators considered include: the combined multiple recursive generator MRG32k3a [6, 31], dynamic creation of Mersene Twisters [28] and the SPRNG Multiplicative Lagged-Fibonacci Generator (MLFG) [24]. For the purpose of comparison we

also test a pLab combined Explicit Inverse Congruential Generator (cEICG) [9, 10]. Their performance is compared from the point of view of their initialization and generation times. Our tests show that initialization can be completed most quickly by MLFG and most slowly by Dynamic Creation. Generation of random numbers was done most quickly by Dynamic Creation's Mersenne Twisters and most slowly by the cEICG.

1 Introduction

Distributed stochastic simulations has become a popular tool for evaluating and testing complex stochastic dynamic systems. However there is some concern about the credibility of the final results of such simulation studies [11]. One of the important issues which need to be properly addressed for ensuring validity of the final results from any simulation study is application of an appropriate source of randomness. In the case of distributed stochastic simulation, the quality of the final results is highly dependent on the underlying parallel Pseudo-Random Number Generator (PRNG).

Parallel PRNGs (PPRNGs) with the required empirical, analytical and deterministic properties are not trivial to find [9, 23, 10]. However, recent research activities have resulted in several generators which we consider to be of high quality [6, 23, 24, 28, 32]. The effectiveness of simulations depends not only on their accuracy but also on their efficiency and so they are also reliant on the speed and flexibility of these PPRNGs.

In this paper, we will examine the required features of modern PPRNGs, from the point of view of their possible applications in Multiple Replications in Parallel (MRIP) paradigm of stochastic simulation [5], in which each processor (as a simulation engine) runs an independent version of the simulation and submits results of measurements (observations) regularly to a central processor for analysis. The MRIP paradigm of simulation has become more popular with emergence of such packages as Akaroa2[1] [4], which supports automatic parallelization of simulated processes.

As MRIP distributes identical replications of a given simulation over different simulation engines, each engine requires an independent sequence of Pseudo-Random Numbers (PRNs) [4].The success of such a simulation is highly dependent on the quality of these multiple independent streams and the efficiency of their generation. This requires that the following properties are observed:

P1. Intra-stream uniformity and independence: It should be impossible to show that the generated numbers $x_1, x_2 \ldots x_p$ cannot be considered as realizations of independent and identically uniformly distributed random variables.

P2. Inter-stream independence: It should be impossible to show that numbers generated by the i^{th} stream $\{x_{i1}, x_{i2}, \ldots\}$ are not independent from those generated in the j^{th} stream $\{x_{j1}, x_{j2}, \ldots\}$ for any i and j.

P3. Satisfactorily many satisfactorily long streams of PRNs: The cycle of each generator used in a given simulation should be sufficiently large to ensure

[1] Akaroa2 [4] is a fully automated simulation tool for running stochastic simulation in MRIP paradigm, developed at the University of Canterbury in Christchurch, New Zealand.

that not whole cycle of PRNs is exhausted within a single application. Furthermore one should be able to generate sufficiently many streams of PRNs (one stream per simulation engine).

P4. Efficient implementation: Initialization of each parallel stream and the subsequent generation of numbers should be efficient in both space(memory) and time.

Given a single stream of PRNs we can apply tests such as those described by Knuth [12] and Marsaglia [20] to ascertain if we are confident that property P1 is satisfied. Being satisfied that we have a generator that is able to produce a single stream of i.i.d random variables which satisfy P1, two paradigms exist for generating parallel i.i.d. streams.

Cycle Splitting is the method of taking a single stream $\{x\}$ of PRNs produced by a single generator and splitting this stream into P sub-streams $\{\{x^1\}, \{x^2\}, \ldots, \{x^P\}\}$, where P is the number of processors/engines used in a given simulation. There are two main variations on this paradigm possible. In *Blocking* we determine a block size B and assign to the i^{th} processor the stream $\{x^i\} = \{x_{iB}, x_{iB+1}, \ldots, x_{iB+(B-1)}\}$. Alternatively, in *Leap-Frog*, if P is the number of processors, we produce the stream for the i^{th} processor as $\{x^i\} = \{x_i, x_{i+P}, x_{i+2P}, \ldots\}$. Both methods involve distributing the finite sequence produced by one generator to P processors. A potential problem is that PRNs generated by linear generators can experience long range correlations [30, 24]. Under cycle splitting such long range correlations can introduce short range inter-stream correlations when using *Blocking*, or short range intra-stream correlations when using *Leap-Frog* [24].

An alternative to *Cycle-Splitting* is *Parameterization*, the method of creating a new, full cycle, independent generator for each processor from a family of generators. *Seed Parameterization* is used with generators that produce independent full length cycles depending on the seed value. That is, we give each processor the same generator but initialized with different seeds so that each processor has access to a different independent stream. *Iteration Function Parameterization* modifies some value within the iteration function so that each processor uses a different generator. The limiting factor here is how many independent streams the parameterization method can produce for a particular generator and how quickly it can produce them.

The required minimum cycle lengths of PRNGs which can satisfy property P3, regardless of computing technology in which they would be implemented, have been considered in [33]. Namely, assuming that (i) Moore's law remains applicable also in post-electronic computers and the frequencies of CPU clocks will continue to double each 1.5 year (or each 2 years, or each 2.5 years, respectively), (ii) in 2000, popular computers were equipped in the CPU operating at 800 MHZ, one can see that clocks of typical CPUs would operate with 100 THz in 2025 (or 2034 or 2042, respectively), see Figure 1. This means that we would use multiple processor computers in all-optical technology, then.

According to recently established theoretical restrictions on the number of pseudo-random numbers, a PRNG generating numbers in a cycle of length L should be used in a single simulation as a source of not more $16\sqrt[3]{L}$ numbers in the case of linear generators [11], and of not more than L numbers in the case of non-linear generators.

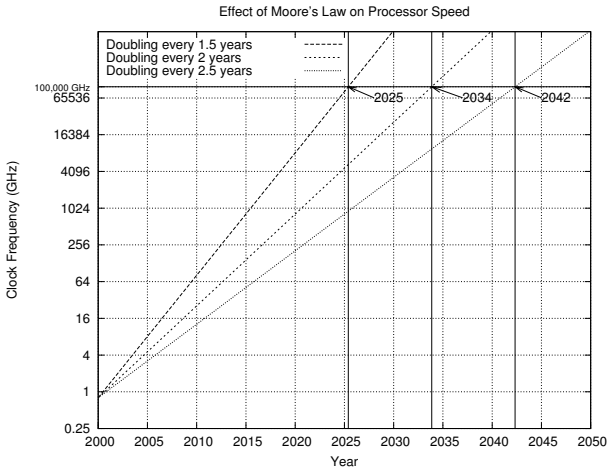

Figure 1: Effect of Moore's Law on CPU's clock frequency

Assuming that only 1% of whole simulation time is spent on generating pseudo-random numbers, an all-optical computer with a CPU clock running at 100THz would need a linear PRNG with the cycle length of about 2^{110}, for executing a simulation lasting one hour, or about 2^{140} for executing a week long simulation; see Figure 2. In the same situations, non-linear PRNGs could generate numbers in cycles of length 2^{43} or 2^{52}, respectively.

If a generator's cycle is split in substreams, needed by multiple simulation engines which operate on, say, run on $2^{14} = 16384$ all-optical processors, then such a linear PRNG should have the cycle length of about 2^{160} for executing a simulation lasting one hour, or about 2^{182} for executing a week long simulation, see Figure 3. Then, a non-linear PRNGs would need to generate numbers in cycles of length 2^{57} or 2^{65}, respectively.

A modern PPRNG must satisfy all the properties described above and several have been proposed that can do so. However, further to being of long period and statistically robust, there must exist its efficient implementation both in terms of memory and speed. It is the purpose of this research to survey modern PPRNGs proposed and test their implementations. In section 2 we introduce the generators that we consider. Section 3 describes our experiments and the results.

2 Parallelizable Generators

As distributed (and parallel) computing has become more available and popular the need for PPRNGs that satisfy the requirements described above has increased. Many PRNGs with parallelization techniques have been proposed. Linear Congruential Generators (LCGs) and Feedback Shift Register Generators have received perhaps the most attention and the mathematical theory and implementations of these generators is subsequently highly developed. The history and properties of two such generators

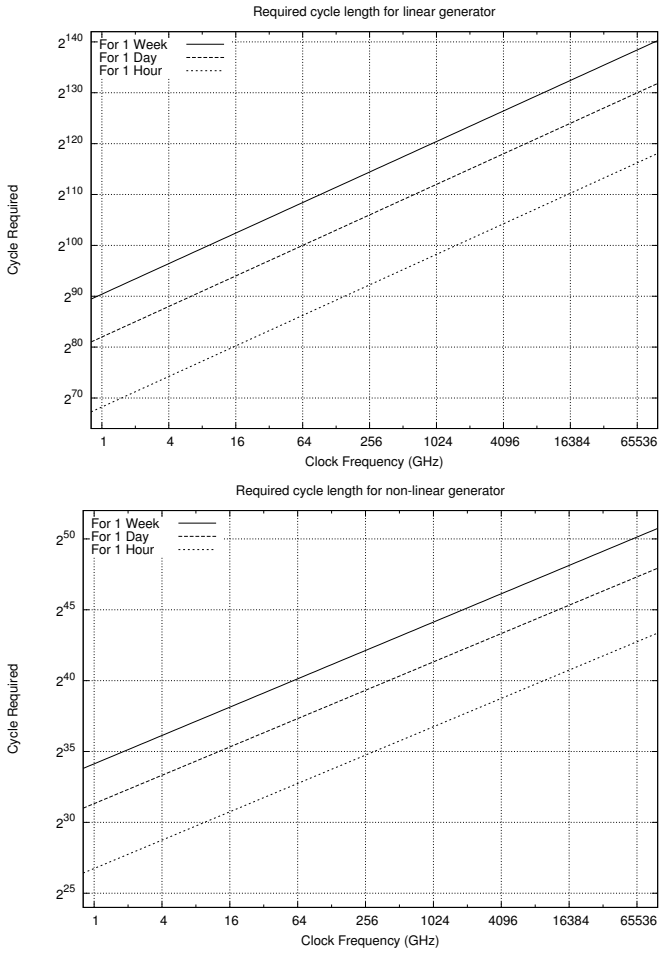

Figure 2: Cycle lengths of PRNGs required for simulations run on single processors

of particular interest to distributed applications are described in sections 2.1 and 2.2. Many alternatives exist however.

Mascagni and Srinivasan[23] describe methods to parameterize the simple linear congruential generator $x_n = ax_{n-1} + b \pmod{m}$ by way of varying a when m is prime, or b when m is a power of 2. The period of such methods is limited by the modulus to $m - 1$ and m respectively. Several disadvantages including poor randomness in the least significant bits make LCGs with $m = 2^k$ a poor choice[23]. The alternative, LCGs with prime moduli, are most often implemented with a Mersenne prime modulus as a fast algorithm exists for modular multiplication with Mersenne primes moduli. However, to find suitable values for a we must know all primitive roots of m, a computationally complex task. To make the calculation of primitive roots trivial, Mascagni and Chi proposed an LCG family with Sophie-Germain prime[2] moduli and a fast modular multiplication algorithm for Sophie-Germain primes[22]. They imple-

[2]Sophie-Germain primes are of the for $m = 2p + 1$ where m and p are prime,

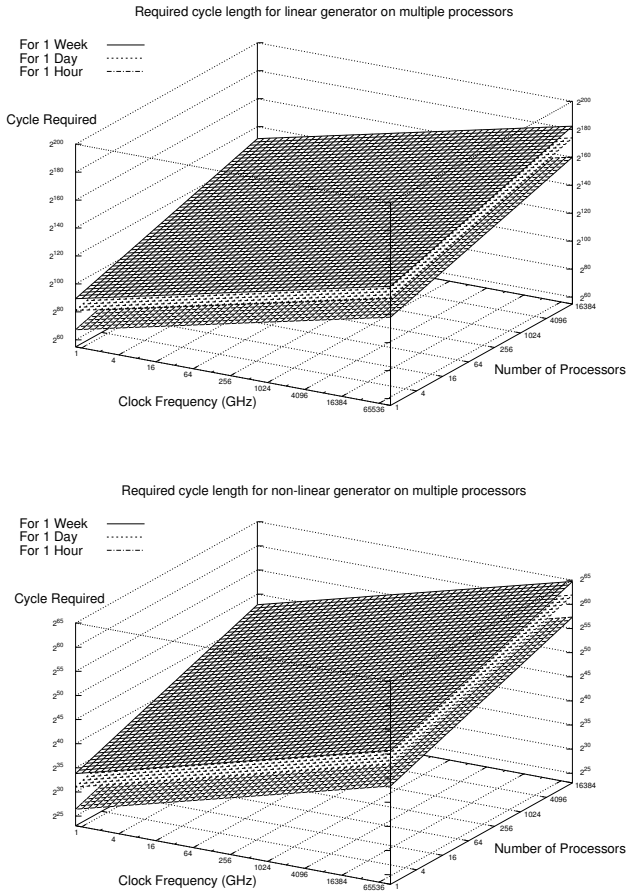

Figure 3: Cycle lengths of PRNGs required for simulations run on multiple processors

mented this method in the form of their Sophie-Germain Modulus Linear Congruential Generator(SGMLCG), a 64-bit LCG family capable of producing up to $2^{63} - 10509$ independent full period generators with period $2^{63} - 21016$. SGMLCG has passed Marsaglia's Diehard tests [20] as well as the tests given as part of Mascagni's SPRNG package [23].

An interesting alternative to linear PRNGs are the Inversive Congruential Generators (ICGs) and Explicit Inversive Congruential Generators (EICGs) by Eichenauer et al.[2, 3]. They have some similar properties to linear congruential generators but have the distinct advantage of the absence of the lattice structure associated with linear generators; see Figure 4. However, a significant disadvantage that has resulted in ICGs and EICGs not being used extensively is that both the recursion for ICGs ($x_n = a\overline{x_{n-1}} + b \pmod p$, $n > 0$) and EICGs ($x_n = \overline{a(n + n_0) + b} \pmod p$, $n \geq 0$) require modular inversion, a costly process. That said the excellent properties of inverse generators suggests they would be very useful in applications where results based

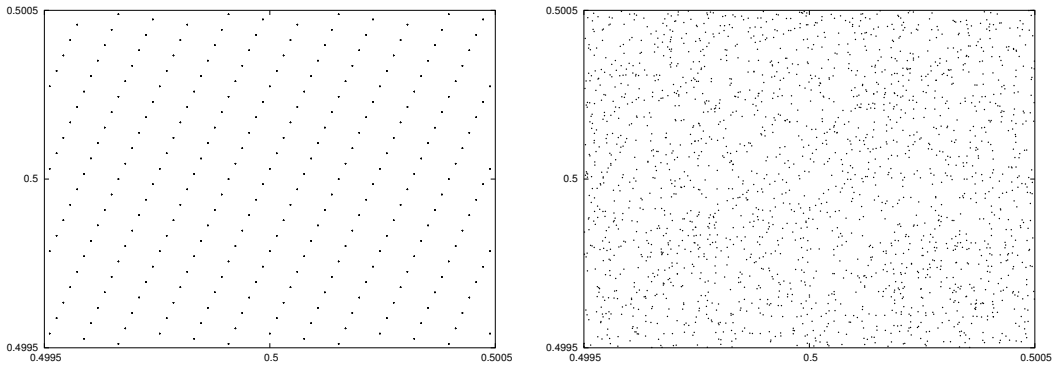

Figure 4: Lattice structure of LCG: $x_n = 65539x_{n-1} \bmod 2^{31}$ (left), and EICG: $x_n = \overline{65539(n+1)} \bmod 2^{31} - 1$ (right).

on linear generators are in doubt. Further study on techniques to make ICGs and EICGs more efficient would be very useful. For further discussion on the theoretical and empirical properties of inverse generators as well as splitting techniques we refer the reader to [10] and [8].

2.1 MRG32k3a - Combined Multiple Recursive Generator

Linear Congruential Generators, first put forward by Lehmer[16] in 1949, are based on the following simple linear recurrence;

$$x_n = ax_{n-1} + b \pmod{m} \quad (1)$$

$$u_n = \frac{x_n}{m} \quad (2)$$

such that u_n is a i.i.d random variable in the range $[0, 1)$.

Early questions on the statistical robustness of LCGs as well their relatively small maximum period of $p = m$ ($p = m - 1$ for $b = 0$) prompted research into the *multiple recursive generator (MRG)*[7], defined as follows;

$$x_n = a_1x_{n-1} + a_2x_{n-2} + \ldots + a_kx_{n-k} \pmod{m} \quad (3)$$

$$u_n = \frac{x_n}{m} \quad (4)$$

When the characteristic polynomial $P(z) = z^k + a_1z^{k-1} + a_2z^{k-2} + \ldots + a_k$ is primitive, the MRG achieves its maximum period of $p = m^k - 1$, a considerable improvement on a LCG with equal modulus. To achieve a high quality in the sense of the *spectral test*, the coefficients of recurrence 3 must be chosen such that $\sum_{i=1}^{k} a_i^2$ is large. However for sake of efficiency we wish to keep these coefficients small. To manage this conflict, L'Ecuyer [15] introduced the combined MRG (CMRG) which combines J MRGs as follows;

$$x_{j,n} = a_{j,1}x_{j,n-1} + \ldots + a_{j,k_j}x_{j,n-k_j} \pmod{m_j} \quad (5)$$

$$u_n = \left(\sum_{j=1}^{J} \frac{x_{j,n}}{m_j} \right) \pmod 1 \quad (j = 1, 2, \ldots, J) \tag{6}$$

L'Ecuyer et al.[15, 6] investigated many parameters to make up the components of this CMRG and has published examples of the CMRGs that are statistically robust, easy to implement, efficient to run and generate PRNs in very long cycles.

An example of such a generator is the so called MRG32k3a which is defined by two MRGs each with three terms as follows;

$$\begin{aligned} a_{11} &= 0 \\ a_{12} &= 1403580 \\ a_{13} &= -810728 \\ m_1 &= 2^{32} - 209 \\ a_{21} &= 527612 \\ a_{22} &= 0 \\ a_{23} &= -1370589 \\ m_2 &= 2^{32} - 22853 \end{aligned}$$

This MRG32k3a has been shown by L'Ecuyer et al.[6] to perform well in statistical tests up to at least 45 dimensions. It has a period of $p \approx 2^{191}$, achievable with arbitrary seed initialization with at least one non-zero element in each MRG.

Being satisfied that MRG32k3a is a strong single stream generator we turn to the task of parallelizing MRG32k3a. An attractive feature of LCGs is that we can easily *fast forward* the generator k steps as follows:

$$x_{n+i} = a^i x_n \pmod m \tag{7}$$

This property allows us to easily perform a similar operation on the MRGs which make up MRG32k3a and parallelize via the *blocking* paradigm[31, 14]. Namely, if we put $s_{j,n}$, the n^{th} state of the j^{th} MRG, as the vector $\{x_{j,n}, x_{j,n+1}, x_{j,n+3}\}$ then a 3×3 matrix A exists such that,

$$s_{j,n+1} = A_j s_{j,n} \pmod{m_j} \tag{8}$$

And so the state $s_{j,n+i}$ is given by,

$$s_{j,n+i} = A_j^i s_{j,n} \pmod{m_j} \tag{9}$$

This approach is taken in the implementation given in [31]. The implementation takes the full cycle and splits it into streams of length 2^{127}, splitting each of those into 2^{51} sub-streams of length 2^{76}. The values 51 and 76 were chosen as $51 + 76 = 127$ and $l = 51, 76, 127$ produced particularly good results with respect to the spectral test performed on numbers from streams starting 2^l places apart.

2.2 Dynamic Creation of Mersenne Twisters

As an alternative to early weak LCGs, the theory of PRNGs based on Feedback Shift Registers (FSRs) was developed. Such generators offered better randomness of numbers than LCGs and, as they worked using only bitwise operations, were very fast. 1973 saw the introduction of the *Generalized FSR (GFSR)* PRNGs [17] which were based on the following recurrence;

$$\vec{x}_{l+n} = \vec{x}_{l+m} \oplus \vec{x}_l, \quad (l = 0, 1, \ldots) \tag{10}$$

where $\vec{x}_i$ is the i^{th} word vector of size w and $\oplus$ is binary addition (exclusive OR). GFSRs were generalized in the sense that, with suitably chosen seed, the period $2^n - 1$ was not reliant on word size of the machine but on n, the number of words used to store the state of the generator, which allowed for arbitrary long periods. Matsumoto and Kurita[26] recognized the merits of GFSR but also identified four disadvantages: 1- selection of seed is difficult, 2- randomness qualities are questionable as it is based on the trinomial $t^n + t^m + 1$, 3- GFSRs period of $2^n - 1$ is much smaller than the theoretical maximum of 2^{nw}, and 4- n words of memory are required to produce a period of $2^n - 1$. To address these disadvantages Matsumoto and Kurita[26] developed the *Twisted GFSR (TGFSR)* PRNG which introduced a twisting matrix $A(w \times w)$ into the GFSR recurrence as follows:

$$\vec{x}_{l+n} = \vec{x}_{l+m} \oplus \vec{x}_l A, \quad (l = 0, 1, \ldots) \tag{11}$$

For appropriately chosen values of n, m and A, TGFSR achieves the maximal period of $2^{nw} - 1$ and, due to the properties of the twisting matrix, achieves better randomness, as the recurrence represents a primitive polynomial with many terms rather than a trinomial. With these advantages such generators were able to be created with periods never before seen, such as the popular T800 with period 2^{800}. Despite these successes, the inclusion of the A matrix in TGFSR introduced a defect in k-distribution for k larger than the order of the recurrence[27]. The difficulty stemmed from trying to set A such that $\vec{x}_l A$ was fast to calculate and to have good k-distribution for large k. To address this difficulty a *tempering* matrix T was introduced as follows;

$$\vec{x}_{l+n} = \vec{x}_{l+m} \oplus \vec{x}_l A, \quad (l = 0, 1, \ldots) \tag{12}$$

$$\vec{z}_{l+n} = \vec{x}_{l+n} T \tag{13}$$

which, for appropriate values for T, is equivalent to using a more computationally complex A. In Eq.12 x_{l+n} is the output which is used in further recursions whereas z_{l+n} is the output random variable.

Testing the characteristic polynomial of A for primitivity requires the complete factorization of $2^{mw} - 1$[35, 26]. For many large nw such decompositions are not known and as such a limited number of large period TGFSRs were possible. To address this limitation Matsumoto and Nishimura[29] invented the Mersenne Twister by adjusting the recurrence 12 to allow for a Mersenne prime period as follows:

$$\vec{x}_{k+n} = \vec{x}_{k+m} \oplus \left(\vec{x}_k^u \mid \vec{x}_{k+1}^l\right) A, \quad (k = 0, 1, \ldots) \tag{14}$$

$$\vec{z}_{k+n} = \vec{x}_{k+n} T \tag{15}$$

such that $nw - r$ is the size of the state array of the generator and 2^{nw-r} is a Mersenne prime. For a predetermined integer $r(0 \leq r \leq w-1)$ the designation $\left(\vec{x}_k^u \mid \vec{x}_{k+1}^l\right)$ means the concatenation of $\vec{x}_k^u$ (the upper $w-r$ bits of $\vec{x}_k$) and $\vec{x}_{k+1}^l$ (the lower r bits of $\vec{x}_{k+1}$).

In the same paper as the Mersenne Twister algorithm, code was released for MT19937, a Mersenne Twister PRNG with a period of $2^{19937}-1$ and 623-dimensional equidistribution up to 32-bit accuracy. Such a massive period is possible as the prime decomposition of a Mersenne prime is trivial and so the testing of primitivity of polynomials becomes much faster[29]. MT19937 has passed empirical tests such as Marsaglia's Diehard tests[20] and Load and Ultimate Load Tests executed by the pLab group [34].

Again, having what we believe to be a high quality generator, we consider how to apply it to create many parallel streams of independent PRNs. No algorithm is currently known for *fast forwarding* MT19937 in a similar way that exists for MRG32k3a, so cycle splitting is not a good option. However, a Mersenne Twister is able to be parallelized through a parameterization technique called Dynamic Creation as described and implemented by Matsumoto and Nishimura in [28]. The implementation of dynamic creation takes parameters such as a unique id (eg process, processor id etc.), word size and a Mersenne prime, and creates a Mersenne twister based on those parameters. The ID is encoded into the characteristic polynomial of the PRNG within the matrix A. Independent IDs ensure relatively prime characteristic polynomials which implies independent streams of parallel PRNs. The published implementation allows creation of up to $2^{\frac{w}{2}}$ parallel Mersenne twisters with periods including Mersenne primes from $2^{521}-1$ to $2^{44497}-1$.

Despite their advantages, Mersenne Twister PRNGs have one notable flaw. The recurrence (14) modifies very few bits at each step and as such a poor distribution in the state array will have long lasting affects in that the poor distribution will remain in the state array for many subsequent states[32]. As such we must take care when initializing the seed of Mersenne Twister as a poor seed may produce a long[3] non-random stream of numbers. This is solved in the implementation of MT19937 and Dynamic Creation by having a LCG to randomly initialize the seed. This is, however, a poor solution to the problem for two reasons. One, we prefer generators that produce i.i.d numbers for any arbitrary choice of seed (other than perhaps all 0's). Two, the massive period of Mersenne Twister is achieved as every permutation of the state array occurs somewhere in the period. As such, even if the seed is not a poor state, the poor states will occur somewhere in the period. Due to the super-astronomical period of Mersenne Twister we are not likely to reach this poor state during any conceivable application so this is a theoretical consideration only. To improve on this problem Panneton and L'Ecuyer have developed WELL generators, an improved long-period generator class based on linear recurrences modulo 2. While these WELL generators perform much better in terms of recovering from a poor state[32], they will still produce a long[4] stream of poorly distributed numbers given a bad state. As such initialization of seed should still

[3]MT19937 initialized with very few 1's in its 19937 bit state array will produce states with significantly less than half of the bits set to 1 for at least 700,000 steps[32]

[4]WELL19937a initialized with very few 1's in its state array will produce states with significantly less than half of the bits set to 1 for at least 700 steps[32]

be done with care and further development of this class of generators to address this problem is needed before arbitrary seed choice is a reality.

2.3 Multiplicative Lagged-Fibonacci Generators

In an effort to achieve larger periods than offered by LCGs, researchers in the 1950s considered recursions which based x_n not only on x_{n-1} but also on x_{n-2} as follows[12]:

$$x_n = ax_{n-1} + cx_{n-2} \pmod{2^b} \tag{16}$$

The period of such a generator, for appropriate values of a, c and $m = 2^b$ is as high as $m^2 = (2^b)(2^b)$, a marked improvement over LCGs with maximum period $m = 2^b$. In the simplest case when $a = c = 1$ the recurrence 16 represents the Fibonacci recurrence which displays poor randomness qualities. To improve on this, a lag l was added to the Fibonacci recurrence as follows[1]:

$$x_n = x_{n-1} + x_{n-l} \pmod{2^b} \tag{17}$$

For large values of l ($l > 15$), the recurrence (17) achieves much better randomness than recurrence (16). Another advantage is that the period, $p = (2^l - 1)(2^{b-1})$, is dependent on l as well as b. As such the period of such a generator can easily be increased by choosing a larger lag l.

Even better performance is achieved supplementing the *lag* l in (17) with a *short lag*, k, as follows,

$$x_n = x_{n-k} + x_{n-l} \pmod{2^b} \tag{18}$$

such that $k < l$. For appropriate values the maximum period of $p = (2^l - 1)(2^{b-1})$ is achieved. This recurrence (18) forms the so called *additive lagged-Fibonacci generator (ALFG)*. The ALFG has been used extensively though it is now recognized that it performs poorly in some relative simple statistical tests for even relative large lags[12]. As such it is necessary to choose l to be very large to ensure a robust generator[24]. An alternative that performs much better is the *multiplicative lagged-Fibonacci generator (MLFG)* defined by the following recurrence,

$$x_n = x_{n-k} \times x_{n-l} \pmod{2^b} \tag{19}$$

Such MLFG demonstrates better robustness than its cousin the ALFG [24], however several features of this generator are noteworthy. Firstly, it has a slightly smaller period that the ALFG, with a maximum period of $p = (2^l - 1)(2^{b-3})$, for appropriate values of k and l and seed $(x_{n-1} \ldots x_{n-l})$. Secondly, due to the multiplicative nature of the generator and the fact that an odd number multiplied with an even number gives an even number, the sequence produced by (19) will eventually become all even, if not all seed values are even. To avoid this transient period at the beginning of the stream, it is recommended to seed a MLFG with all odd values. Further to this, the user must recognize that the least significant bit of all numbers produced by MLFGs seeded in this way will always be 1[24].

In terms of parallelization, cycle splitting and parameterization algorithms exist for both the ALFGs and MLFGs. Efficient cycle splitting via blocking for both ALFGs and MLFGs have been presented by Makino[19]. Mascagni et al.[18] proposed a parallelization of ALFGs based on seed parameterization capable of yielding $2^{(b-1)(l-1)}$ distinct maximum period cycles. A similar technique for MLFGs was proposed by Mascagni and Srinivasan[24] yielding $2^{(b-3)(l-1)}$ uncorrelated cycles. The default lag in Mascagni implementation is $l = 17$ with $b = 64$, giving 2^{976} streams, each of period approximately 3×2^{76}. Similar to their sequential versions, the parameterized MLFGs display better robustness than the parameterized ALFGs with the latter failing some standard tests due to inter-stream and intra-stream correlations. The MLFGs perform well in tests, however, even with small lags[24].

3 Empirical Comparison

We recognize that cycle splitting with MRG32k3a, Dynamic Creation of Mersenne Twisters (if appropriately initialized) and seed parameterization with MLFGs produce high quality parallel streams in terms of inter-stream and intra-stream independence, period length and number of possible streams. However, we wish to consider the efficiency of these generators' published implementations with respect to their applications in MRIP paradigm of stochastic simulation. For this purpose we consider speed of initialization and generation of PRNs in the case of these three methods.

3.1 Platform

Experiments were conducted on a Intel Pentium 4 CPU running at 2.4GHz with 512KB of cache, a floating point unit and 512MB of RAM, running Linux version 2.4.20-24.9, gcc version 3.2.2 and Red Hat 9. As the generators were tested for their use in practical applications, it is unrealistic to expect that the majority of users would re-implement these generators themselves. As such published implementations of the generators were used. A C++ object oriented implementation of MRG32k3a is made available by L'Ecuyer at [13]. C implementation of Dynamic Creation is made available by Matsumoto at [25]. SPRNG[21] (Scalable Pseudo-Random Number Generators) is a library of tested parameterizable generators by Mascagni, which includes the MLFG used here. For the purpose of comparison we include in our tests a combined EICG (cEICG) consisting of three EICGs[5] with a total period of $P = 2^{93}$ and implemented through the pLab's PRNG library[34].

3.2 Initialization

Before taking part in a simulation each engine must be assigned an independent stream of PRNs. We consider initialization to be this process of assigning a stream to an engine, including any calculation of parameters, initialization of seeds etc. Initialization is the

[5]The cEICG used in tests was made up of eicg$(2147483647, 7, 1, 0)$, eicg$(2147483629, 11, 1, 0)$, eicg$(2147483587, 13, 1, 0)$ where these definitions are of the form eicg(p, a, b, n_0)

time taken from the instant when an engine is requesting a stream of PRNs to the point at which it is able to start generating PRNs. The upper plot of Figure (5) shows a comparison between the four generators we consider. Due to the fast constant initialization time of the MLFG a detail of it is shown in the right plot. For each generator we initialize n streams, for various n, and plot the average initialization time for that n. For example if $n = 3$ and a given PPRNG finishes initializing the first stream after 2 seconds, the second after 4 seconds and the third after 6 seconds, the mean initialization time for that generator at $n = 3$ would equal 4. As each engine may begin simulation as soon as its stream is initialized, the mean time is most appropriate for comparison. Initialization of streams for cEICG and MRG32k3a is done

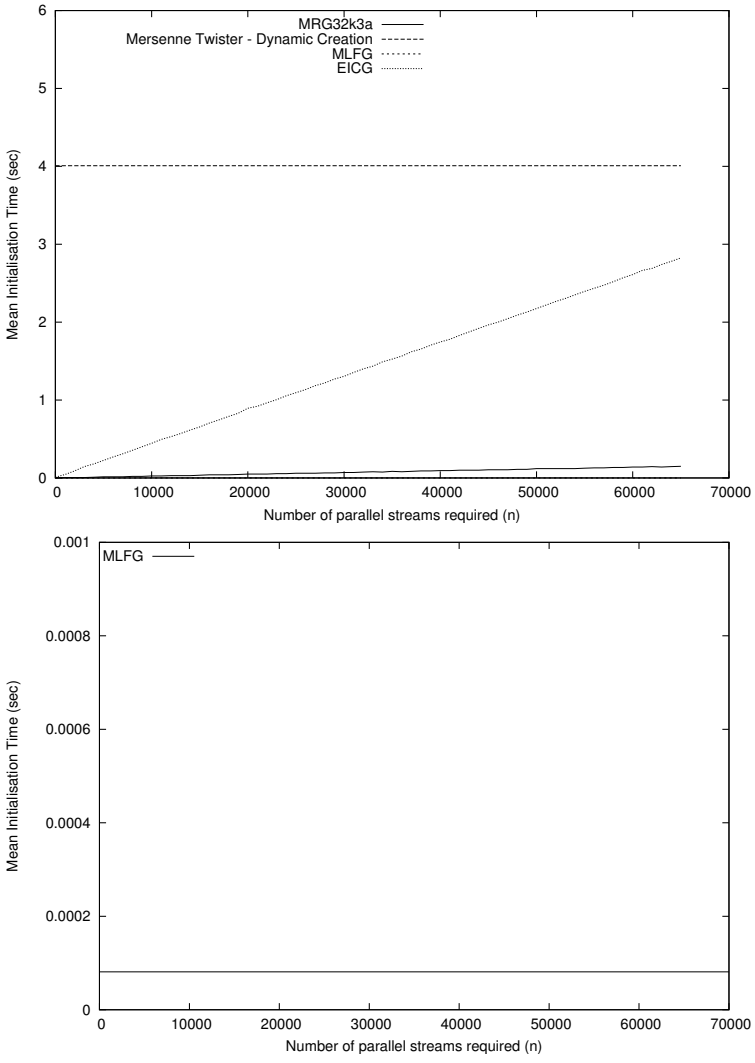

Figure 5: Mean waiting time for stream initialization

using a technique of fast forward and as such is sequential. This means as n becomes

large the average time needed to initialize a stream increases linearly. While both processes are linear, we see the initialization by cEICG is slower than MRG32k3a. The implementation of MRG32k3a assumes fixed stream sizes and as such has precomputed the matrix required to move from one stream to the next. The pLab's cEICG allows for arbitrary stream sizes so no precomputation is possible and so initialization is much slower. Both Dynamic Creation and the SPRNG MLFG follow the parameterization paradigm and both implementations have initialization routines that accept an ID and return independent streams for independent IDs. Working on the assumption that each engine has access to a unique ID, this allows the initialization of Dynamic Creation and SPRNG MLFG to be parallelized in such a way that each engine may initialize it's own generator concurrently without any inter-engine or engine-control unit communication and without fear of loss of independence. As such the average initialization time of these generators is constant as n increases. To reach a figure for these constants 100 Mersenne Twisters and 100,000 MFLG streams were created and the mean time was taken.

MRG32k3a stream were initialized to be the default length of 2^{127}. Dynamic Creation was asked to create Mersenne Twisters with 32 bit word length and period 2^{521}, an implementation minimum. SPRNG created MLFGs with the default lag of $l = 17$ and $b = 64$, yielding generators of period $p \approx 3 \times 2^{76}$. PLab's PRNG library was passed the parameters in Footnote (5) and asked to construct streams of length $p \approx 2^{62}$.

3.3 Generation

We have assumed that once initialization is complete each engine will have access to an independent stream of PRNs of at least 2^{76} numbers in length. As such it is *practically*[6] impossible for an engine to exhaust its allocated stream and require another. So having looked at initialization speed we need only still look at the speed at which the respective PRNGs generate numbers.

All four generators were required to generate n numbers. To encourage an even playing field, all generators were expected to generate numbers in the range $[0, 1)$. The upper plot of Figure (6) shows generation times for all four generators tested. Due to the bunching of the fastest three generators caused by the slowness of the cEICG, a detail of only MRG32k3a, Dynamic Creation and MLFG is shown in the right plot of Figure (6). As expected all generators run in linear time with respect to n. As all operations in Dynamic Creation's Mersenne Twisters are bitwise, it is the fastest of all tested generators. A single multiplication and modulo operations mod 2 make MLFG the second fastest. The more complex operations of MRG32k3a make it the third fastest, while the inversion operation of the cEICG make it by far the slowest. The performance of the cEICG can be improved by reducing the number of EICGs that make up the cEICG, however this will reduce to period of the generator also.

[6] a Pentium 4 running continuously would take approximately 1673 millenia to iterate around a near empty while loop 2^{76} times.

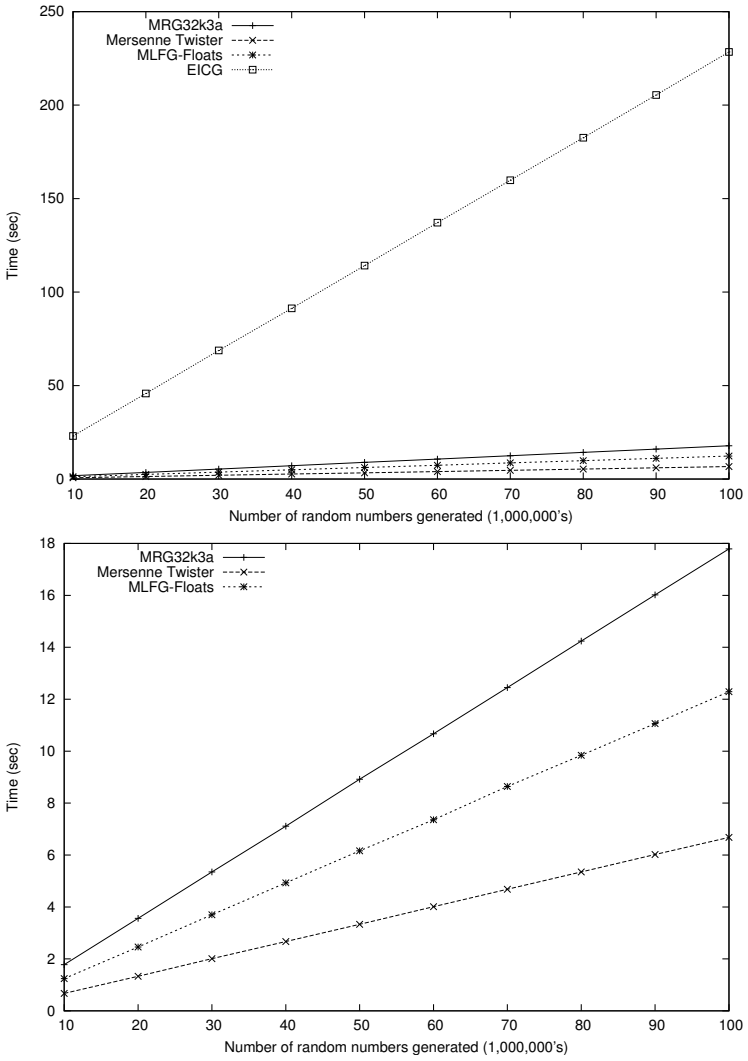

Figure 6: Generation Time per Simulation Engine

4 Conclusion

Executing stochastic simulation in MRIP paradigm has many advantages. However the quality of the final simulation results relies heavily on the quality of the underlying Parallel Pseudo-Random Number Generator (PPRNG). To accept a generator to be of high quality we require that it demonstrate intra-stream uniformity and independence, inter-stream independence, a satisfactorily large period and be able to produce a satisfactorily large number of streams. Further to this we require that an efficient and accurate implementation exists.

Given the existence of a single stream PRNG two paradigms exist which may be employed to achieve parallel streams of PRNs. Cycle splitting, which, as it's name suggests, involves distributing the single large cycle produced by the PRNG among

various streams. Alternatively, parameterization involves modifying some parameter in the base generator such that different parameters result in different independent full period streams.

Though high quality generators are difficult to find, several have been proposed. We investigate several that are potentially useful in massively parallel stochastic simulations in the MRIP scenario. L'Ecuyer's MRG32k3a PRNG is a large period linear generator. It achieves its large period and good randomness by combining several Multiple Recursive Generators and is parallelized by the cycle splitting paradigm. Matsumoto and Nishimura's Dynamic Creation generates independent Mersenne Twister PRNGs by parameterizing the matrix A in recurrence (14). Mascagni and Srinivasan's parameterization of the Multiplicative Lagged-Fibonacci Generator is implemented within the SPRNG library and is based on seed parameterization. We also consider the Explicit Inversive Congruential Generators as implemented within the pLab's PRNG library. However, despite excellent randomness properties, the last class of PRNGs is significantly slower than the other three generators considered.

The generators were tested for initialization and generation speed to assess the efficiency of current implementations. Initialization was completed most quickly by the Multiplicative lagged-Fibonacci Generator and most slowly by Dynamic Creation. Generation of numbers was performed most quickly by Dynamic Creation's Mersenne Twisters and most slowly by the Explicit Inversive Congruential Generator.

5 Acknowledgments

This work was supported by the University of Canterbury, New Zealand, Summer Scholarship (U1042). The author(s) wish to thank Makoto Matsumoto and Pierre L'Ecuyer for their correspondence assisting the writing of this report. The first author would like to thank his parents for their encouragement and support and Jennifer for her inspiration and understanding.

References

[1] J. E. K. Smith, B. F. Green Jr. and L. Klem. Empirical tests of an additive random number generator. *J. ACM*, 6(4):527–537, 1959.

[2] J. Eichenauer and J. Lehn. A non-linear congruential pseudo random number generator. *Statist. Papers*, 27:315–326, 1986.

[3] J. Eichenauer-Herrmann. Statistical independence of a new class of inversive congruential pseudorandom numbers. *Math. Comp.*, 60:375–384, 1993.

[4] K. Pawlikowski, G. C. Ewing and D. McNickle. Akaroa-2: Exploiting network computing by distributing stochastic simulation. In *Proceedings of European Simulation Multiconference ESM'99*, pages 175–181, Warsaw, Poland, 1999. Int. Society of Computer Simulation.

[5] D. McNickle, G. Ewing and K. Pawlikowski. Multiple replications in parallel: Distributed generation of data for speeding up quantitative stochastic simulation. In

Proceedings of 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, pages 379–402, Berlin, August 1997.

[6] D. Ariely, G. Zauberman, G. W. Fischer, Z. Carmon and P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Oper. Res.*, 47(1):159–164, 1999.

[7] A. Grube. Mehrfach rekursiv-erzeugte pseudo-zufallszahlen. *Zeitschrift für angewandte Mathematik und Mechanik*, 53:T223–T225, 1973.

[8] P. Hellekalek. Inversive pseudorandom number generators: concepts, results, and links. In C. Alexopoulos, K. Kang, W.R. Lilegdon, and D. Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference*, pages 255–262, 1995.

[9] P. Hellekalek. Good random number generators are (not so) easy to find. In *Selected papers from the 2nd IMACS symposium on Mathematical modelling—2nd MATHMOD*, pages 485–505. Elsevier Science Publishers B. V., 1998.

[10] A. Uhl, K. Entacher and S. Wegenkittl. Linear and inversive pseudorandom numbers for parallel and distributed simulation. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation*, pages 90–97. IEEE Computer Society, 1998.

[11] K. Pawlikowski, H. J. Jeong and J. R. Lee. Credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine*, 40(1):132–139, 2002.

[12] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[13] P. L'Ecuyer. http://www.iro.umontreal.ca/ lecuyer/.

[14] P. L'Ecuyer. Random numbers for simulation. *Commun. ACM*, 33(10):85–97, 1990.

[15] P. L'Ecuyer. Combined multiple recursive generators. *Operations Research*, 44(5):816–822, 1996.

[16] D. H. Lehmer. Mathematcial methods in large-scale computing units. In *Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery*, pages 141–146. Harvard University Press, Cambridge, Mass., 1951.

[17] T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithm. *J. ACM*, 20(3):456–468, 1973.

[18] D. V. Pryor, M. Mascagni, S. A. Cuccaro and M. L. Robinson. A fast, high quality, and reproducible parallel lagged-fibonacci pseudorandom number generator. *Journal of Computational Physics*, 119:211–219, July 1995.

[19] J. Makino. Lagged-fibonacci random number generators on parallel computers. *Parallel Computing*, 20:1357–1367, September 1994.

[20] G. Marsaglia. Diehard software package. ftp://stat.fsu.edu/pub/diehard.

[21] M. Mascagni. The scalable parallel random number generators library (sprng) for asci monte carlo computations. http://sprng.cs.fsu.edu/.

[22] M. Mascagni and H. Chi. Parallel linear congruential generators with sophie-germain moduli. *Parallel Computing*, 30:1217–1231, November 2004.

[23] M. Mascagni and A. Srinivasan. Sprng: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26(3):436–461, September 2000.

[24] M. Mascagni and A. Srinivasan. Parameterizing parallel multiplicative lagged-fibonacci generators. *Parallel Computing*, 30:899–916, 2004.

[25] M. Matsumoto. http://www.math.sci.hiroshima-u.ac.jp/ m-mat/eindex.html.

[26] M. Matsumoto and Y. Kurita. Twisted gfsr generators. *ACM Trans. Model. Comput. Simul.*, 2(3):179–194, 1992.

[27] M. Matsumoto and Y. Kurita. Twisted gfsr generators ii. *ACM Trans. Model. Comput. Simul.*, 4(3):254–266, 1994.

[28] M. Matsumoto and T. Nishimura. Dynamic creation of pseudorandom number generators. In H.Niederreiter and J.Spanier, editors, *Monte Carlo and Quasi-Monte Carlo Methods*, pages 56–69, 1998.

[29] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[30] A. De Matteis and S. Pagnutti. Parallelization of random number generators and long-range correlations. *Numerische Mathematik*, 53:595–608, 1988.

[31] E. Chen, P. L'Ecuyer, R. Simard and W. Kelton. An object-oriented randomnumber package with many long streams and substreams, 2001.

[32] F. Panneton and P. L'Ecuyer. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software - To Appear*, 2005.

[33] K. Pawlikowski. Pseudo-random number generators for the 21st century. *To be submitted*, 2005.

[34] pLab. http://random.mat.sbg.ac.at/.

[35] N. Zierler and J. Brillhart. On primitive trinomials (mod 2). *Inf. Control*, 13:541–554, 1968.